MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 856 November, 1985

THE COMPUTATIONAL COMPLEXITY OF TWO-LEVEL MORPHOLOGY

G. Edward Barton, Jr.

ABSTRACT:

Morphological analysis requires knowledge of the stems, affixes, combinatory patterns, and spelling-change processes of a language. The computational difficulty of the task can be clarified by investigating the computational characteristics of specific models of morphological processing. The use of finite-state machinery in the "two-level" model by Kimmo Koskenniemi gives it the appearance of computational efficiency, but closer examination shows the model does not guarantee efficient processing. Reductions of the satisfiability problem show that finding the proper lexical-surface correspondence in a two-level generation or recognition problem can be computationally difficult. However, another source of complexity in the existing algorithms can be sharply reduced by changing the implementation of the dictionary component. A merged dictionary with bit-vectors reduces the number of choices among alternative dictionary subdivisions by allowing several subdivisions to be searched at once.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. A version of this paper was presented to the Workshop on Finite-State Morphology, Center for the Study of Language and Information, Stanford University, July 29-30, 1985; the author is grateful to Lauri Karttunen for making that presentation possible. Bob Berwick has provided useful guidance and commentary in this research, and the paper has also been improved in response to suggestions from Bonnie Dorr and Eric Grimson.

1. Introduction

The "dictionary lookup" stage in a sophisticated natural-language system can involve much more than simple information retrieval. In text, the words that the system knows may show up in heavily disguised form. Inflectional endings such as tense and plural markings may be present; the addition of prefixes and suffixes may change part-of-speech and meaning in systematic ways; in many languages words may have unrelated clitics attached. The addition of prefixes, suffixes, and endings is often accompanied by spelling changes as well; in English, `try+s` becomes `tries` and `dig+er` becomes `digger`. The rules of spelling change can be rather complex.

Superficially, it seems that word recognition might potentially be complicated and difficult. This paper examines the question more formally by investigating the computational characteristics of the "two-level" model of morphological processes (§2). Given the kinds of constraints that can be encoded in the model, how difficult can it be to translate between lexical and surface forms? Although the use of finite-state machinery in the two-level model gives it the appearance of computational efficiency, the model itself does not guarantee efficient processing. Taking the KIMMO system (Karttunen, 1983) for concreteness, sections 4 and 6 will show that the general problem of mapping between lexical and surface forms in two-level systems is computationally difficult in the worst case. If null characters are excluded, the problem is NP-complete. If null characters are completely unrestricted, the problem is PSPACE-complete and thus probably even harder in the worst case. The fundamental difficulty of the problems does not seem to be a precompilation effect (§5).

1.1. Morphological analysis

The word-level processing carried out by a natural-language system is formally a type of *morphological analysis,* concerned with recovering the internal structures of input words. For example, `singing` can be recognized as an inflected form of the verb `sing`, while `unhappy` can be analyzed as `un+happy`. However, the morphological component cannot break words up blindly; despite appearances, `duckling` is not the `-ing` form of a verb. The morphological analyzer must know the basic words of the language in addition to the prefixes and suffixes. In fact, analysis must be guided by more specific constraints as well. Not every word can combine with every affix; it would be an error to analyze `unit` as `un+it` or `beer` as `be+er` (compare `doer`).

The number of inflected forms of a given word is smaller in English than in many other languages. As a result, for a system with small scope it often suffices to trivialize morphological analysis by listing all inflected forms in the dictionary directly. The trivial approach is not feasible for heavily inflected languages such as Finnish, in which a word can have thousands of possible forms. In such cases, both practicality and elegance require a more systematic treatment in terms of inflectional endings, mood and tense markers, clitics, and so forth.

The problem of recovering the internal structures of words can take an extreme form in languages that allow productive compounding. Kay and Kaplan (1982) illustrate such a situation with the German word `Lebensversicherungsgesellschaftsangestellter`, which means *life insurance company employee.* An exhaustive dictionary is impractical when such free compounding is possible.

1.2. Spelling changes

Besides knowing the stems, affixes, and co-occurrence restrictions of a language, a successful morphological analyzer must take into account the *spelling changes* that often accompany the addition of suffixes and similar elements.[1] The program must expect `love+ing` to appear as `loving`, `fly+s` as `flies`, `lie+ing` as `lying`, and `big+er` as `bigger`. Its knowledge must be sufficiently sophisticated to distinguish such surface forms as hopped (= `hop+ed`) and hoped (= `hope+ed`). Cross-linguistically, spelling-change processes may span either a limited or a more extended range of characters (§1.2.1), and the material that triggers a change may occur either before or after the character that is affected (§1.2.2). Complex copying processes (§1.2.4) may be found in addition to simpler, more specific changes.

1.2.1. Local and long-distance processes

The spelling changes associated with the addition of English suffixes are *local* in the sense that they do not affect letters far away from the word–suffix boundary. However, there are processes in other languages that operate over longer distances. The spelling of Turkish suffixes is systematically affected by *vowel harmony* processes, which require the vowels in a word to agree in certain respects.[2] The vowels that appear in a typical suffix are not completely determined by the suffix, but are determined in part by the rules of vowel harmony. The suffix that Underhill (1976) writes as `-sInIz` may appear in an actual word as `-siniz`, `-sunuz`, `-sünüz`, or `-sınız` depending on the preceding vowel. Turkish words may contain large numbers of suffixes, and the effects of vowel harmony can propagate for long distances. (Hungarian suffixes display similar changes.)

1.2.2. Left and right context

Local spelling changes often depend on right context as well as left context; for instance, `carry+ed` changes `y` to `i` but `carry+ing` retains `y`. Less commonly, long-distance changes can also be triggered by material to the right.[3] Verb stems in the Australian language Warlpiri display a regressive change of `i` to `u` triggered by a tense suffix containing a nasal `u`; thus the imperative form of *throw* is `kiji-ka`, but the past-tense form is `kuju-rnu` (Nash, 1980:84). As illustrated, this harmony process can affect more than one `i` in the verb stem. It can also propagate through the element `-rni` that can appear between the verb stem and the tense

[1] Spelling-change processes actually represent a superficial amalgam of phonological changes and orthographic conventions. In this paper, these two aspects of spelling changes will not be distinguished. The phonology and the orthography of a language do not have the same status for linguistics, but the differences are not relevant for present purposes. Note also that it is the surface spelling of a word that will be presented to a program that analyzes written text.

[2] For details of this process, see Underhill (1976), Clements and Sezer (1982), and numerous references cited therein.

[3] Many current analyses of vowel harmony take it to be a fundamentally nondirectional process, even in languages in which it always appears to operate from left to right. For example, it appears as though the influence of root vowels on affix vowels always proceeds from left to right in Turkish, but this is because Turkish lacks prefixes. Clements and Sezer (1982:246ff) discuss a process of colloquial Turkish in which a vowel is inserted between the initial letters of certain words. The choice of vowel is determined by the usual harmony rules of Turkish, but operating from right to left in this case. See also Poser (1982).

ending. (Warlpiri also has another long-distance harmony process, which operates from right to left.)

Other languages provide further examples of long-distance changes that are conditioned by material to the right. Kay and Kaplan (1982) mention a vowel-change process in Icelandic that causes vowels in the middle of a word to depend on the vowels in a following suffix. The inflectional system of German also involves vowel changes. Poser (1982:131ff) discusses an extreme example of long-distance right-to-left harmony that occurs in the language Chumash. The process that he describes changes s to š throughout the entire word when an š occurs in a suffix; thus s+lu+sisin+waš (*3+all+grow awry+past*) becomes šluši šinwaš (*it is all grown awry*).

1.2.3. Right context and processing ambiguity

The existence of changes that depend on right context implies that the lexical-surface correspondence for a particular character cannot always be determined when the character is first seen in a left-to-right scan. However, right context is not crucial for the occurrence of this difficulty. The same kind of *local ambiguity* can arise even when spelling changes do not depend on right context.

Suppose we were to remove the dependence of the y-to-i change on right context by considering a rule system in which y always changes to i after p.[4] There could still be uncertainty about how analysis should proceed. A surface string beginning spi... could correspond to a lexical string spy... as in spies, but it could equally well correspond to spi... as in spider or spiel. In general, analysis may proceed several characters beyond a choice point before it becomes apparent which choice is correct. This is especially true with a large system vocabulary: in the above example, a system that did not know any spi... words could immediately rule out spi... in favor of spy..., but a system with more complete coverage would have to look further into the input before it could identify the correct choice.

1.2.4. Reduplication

Some languages display a kind of change called *reduplication* that often does not lend itself to analysis by the kinds of mechanisms that are appropriate for the other processes that have been mentioned here. Reduplication processes involve the copying of consonants, vowels, syllables, roots, or other subunits of words. Nash (1980:136ff) describes a reduplication process in Warlpiri that copies the first two syllables of a verb and has various semantic effects. For example, he cites the sentence

pirli	ka	parnta-parnta-rri-nja-mpa		ya-ni
hill	PRES	crouch-REDUP	INF-across	go-NONPAST

The mountain extends in a series of humps.

[4] If y always changes to i after p, what justification could there be for saying that spy and not spi is the correct underlying form? In this trivial constructed example, there is none. In an actual language, there could be evidence from a variety of sources: suffixes beginning with y; harmony processes; rules that create or destroy the p that triggers the change; rules that are triggered by the y before it changes; and so forth.

in which the verb stem parntarri- has undergone reduplication.[5] Lieber's (1980:234ff) discussion of several reduplication processes in the language Tagalog provides other examples. One Tagalog reduplication process copies the first consonant and vowel of the stem, making the copied vowel short; another is similar, but makes the copied vowel long; a third process copies the first syllable and part or all of the second, lengthening the copied vowel of the second syllable. See also McCarthy's (1982:193f) treatment of reduplication in Classical Arabic.[6]

[5]The hyphens in the Warlpiri examples are inserted as an analytical aid for the reader, and do not conform to the standard orthography (Hale, 1982:222).

[6]McCarthy's treatment of Arabic is of theoretical interest for at least two reasons: it helps illuminate the nature of linguistic representations, and it shows a way to derive many characteristics of Arabic reduplication from universal linguistic principles rather than language-particular stipulations.

2. Two-Level Morphology

Given a description of the root forms, the combinatory patterns, and the spelling-change rules of a language, the morphological analysis task is well-defined in an abstract sense. However, a practical morphological analyzer also needs an efficient way of putting its linguistic knowledge to use in actual processing. The KIMMO system described by Karttunen (1983) is attractive for this purpose. KIMMO is an implementation of the "two-level" model of morphological analysis that Kimmo Koskenniemi proposed and developed in his Ph.D. thesis.[7] Spelling-change rules are encoded in a finite-state *automaton component,* while roots and affixes are listed with their co-occurrence restrictions in a *dictionary component.* The focus here is on the automaton component. (Reduplication processes find no easy treatment in the KIMMO system, and will henceforth be ignored.)

2.1. The Automaton Component

The two-level model is concerned with the representation of a word at two distinct levels, the *lexical* or dictionary level and the *surface* level. At the surface level, words are represented as they might show up in text. At the lexical level, words consist of sequences of stems, affixes, diacritics, and boundary markers that have been pasted together without spelling changes. Thus Karttunen and Wittenburg (1983) represent the surface form `tries` as `try+s` at the lexical level. Similarly, the Warlpiri surface form `kijika` might be represented at the lexical level as `kIjI-ka`, where `I` is a special lexical character that can surface as either `i` or `u` according to harmony rules.

2.1.1. Expressing Spelling Changes as Two-Level Automata

A spelling-change rule in the two-level model is expressed as a constraint on the correspondence between lexical and surface strings. For example, consider a simplified "Y-Change" process that changes `y` to `i` before adding `es`. Y-Change can be expressed in the two-level model as a constraint on the appearance of the lexical–surface pairs `y/y` and `y/i`. Lexical `y` must correspond to surface `i` rather than surface `y` when it occurs before lexical `+s`, which will itself come out as surface `es` due to the operation of other constraints.

Each constraint is encoded as a finite-state machine with two scanning heads that move along the lexical and surface strings in parallel. The machine starts out in state `1`, and at each step of its operation, it changes state based on its current state and the pair of characters it is scanning. The automaton that encodes the Y-Change constraint would be described by the

[7] University of Helsinki, Finland, *circa* Fall 1983.

following state table:

"Y-Change" 5 5						
	y	y	+	s	=	*(lexical characters)*
	i	y	=	s	=	*(surface characters)*
state 1:	2	4	1	1	1	*(normal state)*
state 2.	0	0	3	0	0	*(require +s)*
state 3.	0	0	0	1	0	*(require s)*
state 4:	2	4	5	1	1	*(forbid +s)*
state 5:	2	4	1	0	1	*(forbid s)*

In this notation, taken from Karttunen (1983) following Koskenniemi, = is a certain kind of wildcard character. The use of : rather than . after the state-number on some lines indicates that the : states are *final states*, which will accept end-of-input. In order to handle insertion or deletion, it is also possible to have a null character 0 on one side of a pair,[8] but the possibility of nulls will not be given full consideration until section 6.

In processing the lexical–surface string pair try+s/tries, the automaton would run through the state sequence 1,1,1,2,3,1 and accept the correspondence. In contrast, with the string pair try+s/tryes it would block on s/s after the state sequence 1,1,1,4,5 because the entry for s/s in state 5 is zero. With the pair try/tri it would not block with any zero entries, but would still reject the pair because it would end up in state 2, which is designated as non-final.

These examples illustrate how the Y-Change automaton implements dependence on the right context +s. The automaton will accept either of the correspondences y/i and y/y, but if it processes the y/i correspondence, it will enter a sequence of states that will ultimately block unless the y/i pair is followed by the appropriate lexical context +s. The right context for a vowel harmony process might seem more difficult to encode because it may be necessary to ignore several intervening consonants, but such a situation actually presents no problem at all. An automaton state can easily ignore irrelevant characters by looping back to itself.

2.1.2. Multiple Spelling–Change Processes

A language will generally exhibit several different spelling-change processes; for example, Karttunen (1983:177) mentions that Koskenniemi's analysis of Finnish uses 21 rules. By and large, these separate processes can be encoded as separate automata in the KIMMO system. In actual processing, the automata that express various spelling-change constraints will all inspect the lexical–surface correspondence in parallel. The correspondence will be accepted only if every automaton accepts it — that is, if it satisfies every constraint.[9] Because the automata are connected in parallel rather than in series, there are no "feeding" relationships between two-level automata.[10] Figure 1 illustrates the parallel arrangement of the KIMMO

[8] The actual KIMMO system of Karttunen (1983) does not allow null characters at the lexical level, but the omission is inessential (Karttunen, p.c.).

[9] If null characters are allowed, the interpretation of "satisfying every constraint" takes on a certain subtlety. See section 6.

[10] It is a theoretical claim of the two-level framework that intermediate levels of representation and "feeding" relationships are not necessary — that two levels suffice, in other words. Series connection of the automata

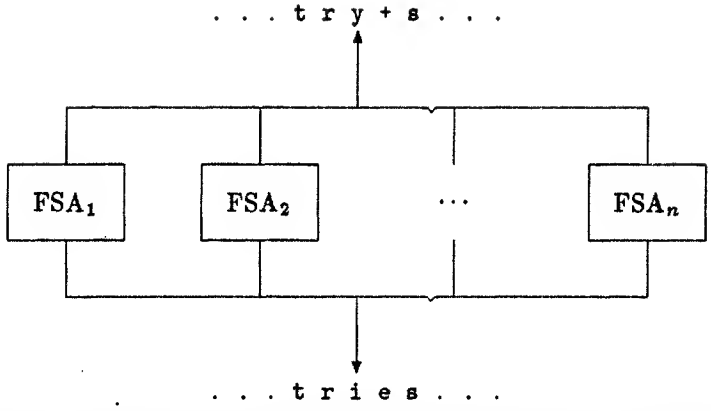


Figure 1: The automaton component of the KIMMO system consists of several two-headed finite-state automata that inspect the lexical-surface correspondence in parallel. Each automaton imposes some constraint on the correspondence. The automata move together from left to right. (From Karttunen, 1983:176.)

automata. A set of several automata can also be compiled into a single large automaton that will run faster than the original set, though its size may be prohibitive (:176f).

2.2. The Dictionary Component

The dictionary component of the KIMMO system is divided into sections called *lexicons*, which are all ultimately reachable from a distinguished *root lexicon*. In the dictionary-level processing for words such as `singing`, KIMMO first locates the lexical form `sing` in the root lexicon. The mechanism for indicating co-occurrence restrictions involves listing a set of continuation lexicons for each entry, and in this case one possibility will be a lexicon that contains `+ing`. In the actual operation of the KIMMO system, dictionary processing is efficiently interleaved with the operation of the automata in such a way that the two components mutually constrain their operations.

The continuation-class mechanism that the KIMMO dictionary uses to encode co-occurrence restrictions among roots and affixes has only finite-state power; each lexicon corresponds to a state in a transition network. As many people have noticed (*e.g.* Karttunen, 1983:180; Karttunen and Wittenburg, 1983:222f), such a design makes it difficult or impossible to express some morphological constraints. In the future, the KIMMO dictionary component will almost

would imply the existence of intermediate representation levels at the interface between automata. Beyond the question of computational efficiency, the theoretical claims of the two-level model will not be evaluated here. Possible arguments against them could involve (a) rule orderings with depth > 1, (b) particular analyses in which the availability of only two levels leads to redundancy in the automata, and (c) multi-part alternative representations (*e.g.* from autosegmental theory) that allow a more illuminating description of various linguistic processes. One possible argument for them could involve the multiplicity of possibilities for rule ordering in a model with intermediate derivational steps.

certainly be redesigned.

The automaton component rather than the dictionary component of the KIMMO system is the main object of attention here, and little more will be said about the dictionary component until section 7.1.

2.3. Generation and Recognition

A KIMMO system does not particularly lean toward either *generating* or *recognizing* the words of a language. Since the machines of the automaton component just express constraints on permissible lexical–surface correspondences, they can serve equally well to determine the lexical form of a surface word (recognition) or to map a lexical stem with affixes into the proper surface form (generation). The only major difference is whether the process is driven by the surface or lexical form. However, the recognition algorithm is slightly more complicated because it uses the lexicon as well as the automata to constrain the analysis of an input word. (As Karttunen (1983:184) notes, it would require only a simple change to run the recognizer without the constraints of the stem lexicon. Such a mode of operation would be useful for stripping recognizable suffixes from unfamiliar roots.)

3. The Seeds of Complexity

The use of finite-state machinery gives the two-level model the appearance of computational efficiency, but in the worst case a KIMMO generator or recognizer has a lot of work to do. This section probes possible sources of complexity, while the next section will exploit them in mathematical reductions that answer the question of how hard KIMMO generation and recognition can be in the general case.

3.1. The Lure of the Finite–State

At first glance, the KIMMO system raises hopes of unfailing efficiency. Both recognition and generation seem to be a matter of stepping finite-state machines through the input from left to right, a process that takes only a quick array reference or so per character. Any nondeterminism that might arise causes little initial concern, since methods of determinizing finite-state machines are well-known. Lexical lookup can also be done quickly, character by character, interleaved with the speedy left-to-right progress of the automata:

> It is a common technique to represent lexicons as letter trees because it minimizes the time spent on searching for the right entry. The recognizer only makes a single left-to-right pass as it homes in on its target in the lexicon. (Karttunen, 1983:178)

The fundamental efficiency of finite-state machines promises to make the speed of KIMMO processing for a language largely independent of the nature of the constraints that the automata encode:

> The most important technical feature of Koskenniemi's and our implementation of the Two-level model is that morphological rules are represented in the processor as automata, more specifically, as finite state transducers .... One important consequence of compiling [the grammar rules into automata] is that the complexity of the linguistic description of a language has no significant effect on the speed at which the forms of that language can be recognized or generated. This is due to the fact that finite state machines are very fast to operate because of their simplicity .... Although Finnish, for example, is morphologically a much more complicated language than English, there is no difference of the same magnitude in the processing times for the two languages .... [This fact] has some psycholinguistic interest because of the common sense observation that we talk about "simple" and "complex" languages but not about "fast" and "slow" ones. (:166f)

In order for the automaton-based two-level model to be of psycholinguistic interest in this way, it must be the model itself that wipes out processing difficulty, rather than some accidental property of the constraints that the automata encode. In much the same vein, Lindstedt (1984:171) remarks following Koskenniemi that "it is psycholinguistically interesting to note that the [two-level] rules are equivalent to such computationally simple and effective [*i.e.* efficient] devices," again picking out the finite-state machinery as the factor responsible for computational efficiency.

3.2. Sample Recognizer Behavior

In assessing the computational characteristics of the KIMMO processing algorithms, it is logical to begin with an example. Figure 2 shows the operations that a KIMMO recognizer for English goes through when it analyzes the word `spiel`. From inspecting the sequence of lexical forms that are considered, it is clear that the recognizer does more than just gliding from left to right through the string.

For example, at step 7 the recognizer is considering the lexical string `spy+`, `y` surfacing as `i` and `+` as `e`, under the theory that the input word might be a plural form of the noun `spy` — `spies` or `spies'`, that is. At step 9 that analysis has failed to pan out and `spy+` is considered again, this time with `+` coming out null on the surface instead of matching the input `e`. At step 11 the recognizer has dropped back to the form `spy` that it was considering at step 4, this time taking the root as a verb. All of the `spy` possibilities ultimately fail, and at step 52 the recognizer finally tries `spi` instead, repudiating the incorrect choice that it made in step 3. In step 53 it assumes that the `e` in the lexical form `spie...` might have been deleted, but this idea soon founders. Finally, in step 59 it finds the correct lexical entry `spiel`.

3.3. Sources of Runtime Complexity

Traces of recognizer operation reveal several factors that combine to determine the overall computational difficulty of an analysis. The recognizer must run the finite-state machines of the automaton component and descend the letter trees that make up a lexicon, it must decide which suffix lexicon to explore after finding a root, and it must discover the correct lexical–surface correspondence.

3.3.1. Stepping through the automata and the lexicon

First of all, some of the recognizer's activities are concerned with the *mechanical operation* of the automata and the letter trees of the lexicon. Running the automata is expected to be fast; there are many well-known fast implementations of finite-state machines, differing somewhat in their time and space requirements. Descending a letter tree should also be easy, in any of its common implementations.

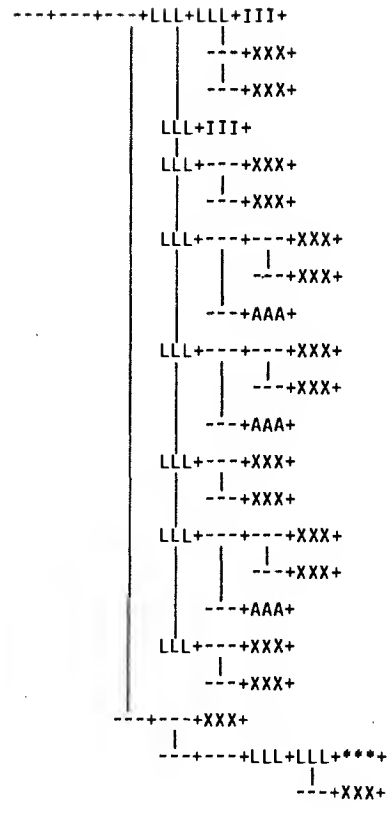
3.3.2. Choosing among alternative lexicons

Second, the recognizer often makes unfortunate choices about the path that it should follow through the collection of lexicons in the dictionary component. Quite a few nodes in the search tree of Figure 2 represent choices among alternative lexicons (LLL). For example, at step 11 the recognizer may search any of several lexicons next: the lexicon `I` that encodes the fact that the present indicative of a verb may have no added ending, the lexicon `AG` that contains the agentive ending `+er`, or one of several other lexicons that contain `+ed` and other inflectional endings.

The search for a path through the suffix lexicons of the dictionary component can take considerable time in the current KIMMO implementation. However, such wandering can be

```
Recognizing surface form "spiel".
  1       s              1,4,1,2,1,1
  2       sp             1,1,1,2,1,1
  3       spy            1,3,4,3,1,1
  4       "spy" ends,    new lexicon N
  6       "0" ends,      new lexicon C1
  6       spy            XXX extra input
  7 (5)   spy+           1,5,16,4,1,1
  8       spy+           XXX
  9 (5)   spy+           1,6,1,4,1,1
 10       spy+           XXX
 11 (4)   "spy" ends,    new lexicon I
 12       spy            XXX extra input
 13 (4)   "spy" ends,    new lexicon P3
 14       spy+           1,6,1,4,1,1
 15       spy+           XXX
 16 (14)  spy+           1,6,16,4,1,1
 17       spy+           XXX
 18 (4)   "spy" ends,    new lexicon PS
 19       spy+           1,6,1,4,1,1
 20       spy+e          1,1,1,1,4,1
 21       spy+e          XXX
 22 (20)  spy+e          1,1,4,1,3,1
 23       spy+e          XXX
 24 (19)  spy+           1,5,16,4,1,1
 25       spy+e          XXX Epenthesis
 26 (4)   "spy" ends,    new lexicon PP
 27       spy+           1,6,1,4,1,1
 28       spy+e          1,1,1,1,4,1
 29       spy+e          XXX
 30 (28)  spy+e          1,1,4,1,3,1
 31       spy+e          XXX
 32 (27)  spy+           1,5,16,4,1,1
 33       spy+e          XXX Epenthesis
 34 (4)   "spy" ends,    new lexicon PR
 35       spy+           1,6,1,4,1,1
 36       spy+           XXX
 37 (35)  spy+           1,6,16,4,1,1
 38       spy+           XXX
 39 (4)   "spy" ends,    new lexicon AG
 40       spy+           1,6,1,4,1,1
 41       spy+e          1,1,1,1,4,1
 42       spy+e          XXX
 43 (41)  spy+e          1,1,4,1,3,1
 44       spy+e          XXX
 45 (40)  spy+           1,6,16,4,1,1
 46       spy+e          XXX Epenthesis
 47 (4)   "spy" ends,    new lexicon A8
 48       spy+           1,6,1,4,1,1
 49       spy+           XXX
 50 (48)  spy+           1,5,16,4,1,1
 51       spy+           XXX
 52 (3)   spi            1,1,4,1,2,5
 53       spie           1,1,16,1,6,1
 54       spie           XXX
 55 (53)  spie           1,1,16,1,6,6
 56       spiel          1,1,16,2,1,1
 57       "spiel" ends,  new lexicon N
 58       "0" ends,      new lexicon C1
 59       "spiel"        *** result
 60 (58)  spiel+         1,1,16,1,1,1
 61       spiel+         XXX

(("spiel" (N SG)))
```




```
Key to tree nodes:

---   normal traversal
LLL   new lexicon
AAA   blocking by automata
XXX   no lexical-surface pairs
      compatible with surface
      char and dictionary
III   blocking by leftover input
***   analysis found
```

Figure 2: These traces show the steps that the KIMMO recognizer for English goes through while analyzing the surface form **spiel**. Each line of the table on the left shows the lexical string and automaton states at the end of a step. If some automaton blocked, the automaton states are replaced by an XXX entry. An XXX entry with no automaton name indicates that the lexical string could not be extended because the surface character and lexical letter tree together ruled out all feasible pairs. After an XXX or *** entry, the recognizer backtracks and picks up from a previous choice point, indicated by the parenthesized step number before the lexical string. The tree on the right depicts the search graphically, reading from left to right and top to bottom with vertical bars linking the choices at each choice point. The figures were generated with a KIMMO implementation written in an augmented version of MACLISP based initially on Karttunen's (1983:182ff) algorithm description; the dictionary and automaton components for English were taken from Karttunen and Wittenburg (1983) with minor changes. This implementation searches depth-first as Karttunen's does, but explores the alternatives at a given depth in a different order from Karttunen's.

```
Recognizing surface form "spiel".
  1       s              1,4,1,2,1,1
  2       sp             1,1,1,2,1,1
  3       spy            1,3,4,3,1,1
  4       "spy" ends,    new lexicon (/N)
  5       "0" ends,      new lexicon (C1)
  6       spy            XXX extra input
  7 (5)   spy+           1,5,16,4,1,1
  8       spy+           XXX
  9 (5)   spy+           1,6,1,4,1,1
 10       spy+           XXX
 11 (4)   "spy" ends,    new lexicon (/V)
 12       spy            XXX extra input
 13  +    spy+           1,6,1,4,1,1
 14       spy+e          1,1,1,1,4,1
 15       spy+e          XXX
 16 (14)  spy+e          1,1,4,1,3,1
 17       spy+e          XXX
 18 (12)  spy+           1,5,16,4,1,1
 19       spy+e          XXX Epenthesis
 20 (3)   spi            1,1,4,1,2,5
 21       spie           1,1,16,1,6,1
 22       spie           XXX
 23 (21)  spie           1,1,16,1,5,6
 24       spiel          1,1,16,2,1,1
 25       "spiel" ends,  new lexicon (/N)
 26       "0" ends,      new lexicon (C1)
 27       "spiel"        *** result
 28 (26)  spiel+         1,1,16,1,1,1
 29       spiel+         XXX

(("spiel" (N SG)))
```

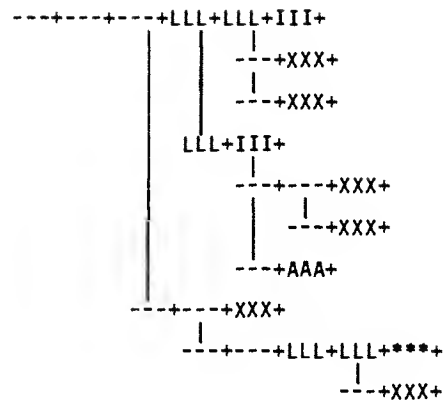


Figure 3: The dictionary modification that will be described in section 7.1 causes the KIMMO recognizer to make fewer choices among lexicons. These traces show the steps that the recognizer goes through in the analysis of spiel when the merged dictionary is used; the number of lexicon-choice nodes (LLL) is lower than in Figure 2. The names of the merged lexicons are written in parenthesized form to indicate that each one actually represents a *class* of lexicons in the original dictionary description. A + entry in the backtracking column indicates backtracking from an immediate failure in the previous step, which does not require the full backtracking mechanism to be invoked.

sharply reduced by merging the lexicons in such a way that several lexicons can be searched in parallel; section 7.1 will explain in detail. Meanwhile, taking this improvement for granted will make it possible to sidestep the problem and focus on other processes. With the merged dictionary, Figure 3 shows that the number of lexicon–choice alternatives in the search tree for spiel is reduced from 8 to 2,[11] cutting the total number of steps from 61 to 29. (The choice between spy–noun and spy–verb remains because it would be directly reflected in the output, but the purely internal choices among the lexicons for different verbal endings are eliminated.)

3.3.3. Finding the lexical–surface correspondence

Finally, some of the backtracking results from local ambiguity in the construction of the *lexical–surface correspondence*. Even if only one possibility is globally compatible with the constraints imposed by the lexicon and the automata, there may not be enough evidence at every point in processing to choose the correct lexical–surface pair; search behavior results.

[11] These figures count LLL nodes excluding unambiguous choices.

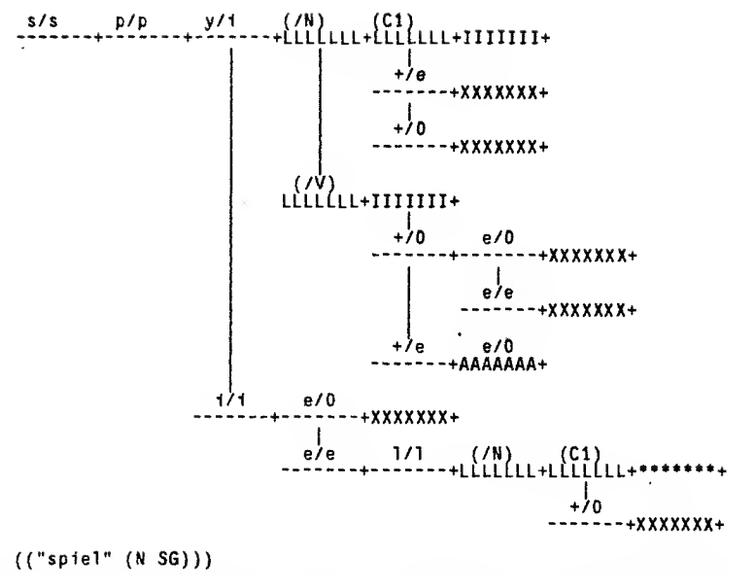


Figure 4: This expanded version of the search tree from Figure 3 shows what hypothesis the KIMMO recognizer is entertaining along each path, during the analysis of `spiel` with a merged dictionary.

Figure 4 displays the search graphically with an expanded version of the merged-lexicon search tree from Figure 3, annotated with information about the specific choices the recognizer has at each point.

Thus, after seeing the surface characters `spi...`, the recognizer did not have enough evidence to choose between the lexical possibilities `spy...` and `spi...`, even though only one analysis was possible for the complete input `spiel`. During exploration of the `spy...` possibility in the `(/V)` lexicon, there was uncertainty about the pairs `+/0`, `+/e`, `e/0`, and `e/e`. It proved unprofitable to explore those regions of the tree in the analysis of `spiel`, but Figures 5 and 6 show that the correct analysis can lie in those regions for other words.

Similarly, in analyzing the word `rubbish` (Figure 7), the recognizer cannot tell after seeing only `rubb...` whether the lexical string is `rubb...` as in `rubbish` or `rub+...` as in `rub+ing ==> rubbing`. In fact, it briefly considers the possibility that surface `r...` might correspond to lexical `re'...` as in the stress-marked lexical representation `re'fer`, but it quickly discovers that the right context for licensing the `e/0` pair is absent. (Recall from section 2.1.1 how a KIMMO automaton implements a change that depends on right context: initially it permits the changed pair in the expectation that the proper right context will be found, and upon processing the changed pair, it enters a state-sequence that will eventually block without the necessary right context.)

In these cases, misguided search subtrees did not get very deep — largely because the relevant spelling-change processes were local in character. Long-distance harmony processes are also possible (§1.2), and thus there can potentially be a long interval before the acceptability of a lexical-surface pair is ultimately determined. For example, when vowel alternations within a verb stem are conditioned by the occurrence of particular tense suffixes, it may be necessary

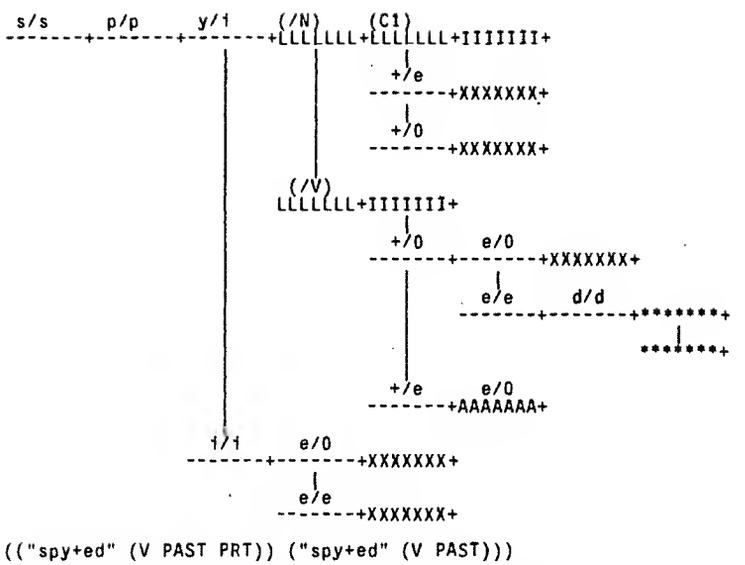


Figure 5: The search tree for **spied** is similar to the search tree for **spiel** (Figure 4), but the solution lies in a different region of the tree. Neither part of the search can be eliminated, since either one may contain the solution.

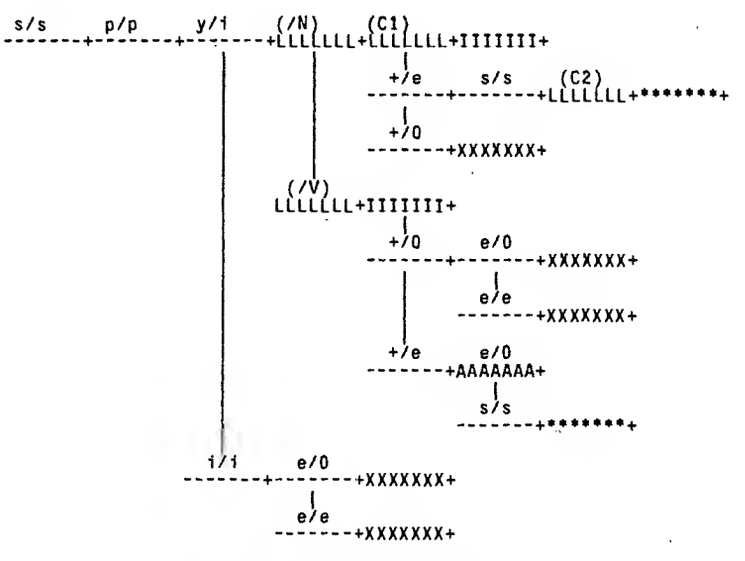


Figure 6: In the analysis of **spies**, the location of the solution in the search tree is different from its location for **spiel** (Figure 4) or **spied** (Figure 5). Thus none of the three main regions of the tree can be pruned from the search.

```
Recognizing surface form "rubbish".
  1      r              1,1,1,2,1,1        12   +   rub+i              1,1,1,1,2,5
  2      re             1,1,1,1,4,1        13       rub+i              XXX
  3      re'            XXX Elision        14  (6)  rubb              1,1,16,2,1,1
  4  (2) ru             1,1,4,1,2,1        15       rubbi              1,1,16,1,2,5
  5      rub            1,1,5,2,1,1        16       rubbis             1,4,16,2,1,1
  6      "rub" ends,    new lexicon (/V)   17       rubbish            1,3,16,2,1,1
  7      rub            XXX extra input    18       "rubbish" ends,    new lexicon (/N)
  8   +  rub+           1,1,3,1,1,1        19       "0" ends,          new lexicon (C1)
  9      rub+e          XXX Gemination     20       "rubbish"          *** result
 10  (7) rub+           1,1,2,1,1,1        21  (19) rubbish+           1,6,16,1,1,1
 11      rub+e          XXX Gemination     22       rubbish+           XXX

(("rubbish" (N SG)))
```

Figure 7: While analyzing the surface form rubbish, the KIMMO recognizer is temporarily misled (i) by the possibility that a lexical e' might have been deleted at the surface and (ii) by the possibility that the surface bb might have resulted from doubling of a single underlying b. However, in each case the possibility fails to pan out. (Refer to Figure 2 for an explanation of the table format.)

to see the end of the word before making final decisions about the stem.[12] The possibility of a long period of uncertainty forms the basis for the reductions in section 4.

3.4. Search and Verification

Setting aside until section 7.1 the problem of choosing among alternative lexicons, it is easy to see that the use of finite-state machinery helps control only one of the two remaining sources of complexity. Stepping the automata should be fast, but the finite-state framework does not guarantee speed in the task of guessing the correct lexical–surface correspondence. The search required to find the correspondence may predominate.

In fact, the KIMMO recognition and generation problems bear an ominous resemblance to problems in the computational class $\mathcal{NP}$. $\mathcal{NP}$ consists of the problems that can be solved on a *N*ondeterministic Turing machine within *P*olynomial time. Informally, a problem in $\mathcal{NP}$ has a solution that may be hard to *guess* (hence the use of nondeterministic machines) but is easy to *verify* (in polynomial time):

> [Informally,] we view [a nondeterministic algorithm] as being composed of two separate stages, the first being a *guessing stage* and the second a *checking stage* .... (Garey and Johnson, 1979:28)
>
> It should be evident that a "polynomial time nondeterministic algorithm" is basically a definitional device for capturing the notion of polynomial time verifiability, rather than a realistic method for solving decision problems. (:29)

This difference in difficulty between guessing and verification seems to fit the KIMMO framework: the finite-state two-level automata can verify a solution quickly, but it may still be hard to guess the correct lexical–surface correspondence.

[12] Since long-distance right context is part of the problem, it has been suggested that KIMMO processing in the problematic cases would be easier if carried out from right to left. However, the more common left context would then cause difficulties, and what could be done about mixed rule systems in which both left and right context play a role? In fact, the reductions in section 4 show that no simple fix will help in the general case.

It is not always apparent from local evidence how to construct a lexical–surface correspondence that will satisfy the constraints imposed by a set of two-level automata: thus the KIMMO algorithms contain the seeds of complexity. The next sections will exploit those seeds in mathematical reductions that prove KIMMO recognition and generation are computationally difficult in the worst case. The finite-state two-level framework itself does not guarantee computational efficiency.

4. The Complexity of Two-Level Morphology

The reductions in this section show that two-level automata can describe computationally difficult problems in a very natural way. It follows that the two-level framework itself cannot guarantee computational efficiency. If the words of natural languages are easy to analyze, the efficiency of processing must result from some additional property that natural languages have, beyond those that are captured in the two-level model.[13] Otherwise, computationally difficult problems might turn up in the two-level automata for some natural language, just as they do in the artificially constructed languages here. In fact, the reductions are abstractly modeled on the KIMMO treatment of harmony processes and other long-distance dependencies in natural languages (see §§3.3.3,1.2).

4.1. The SAT Problem

The reductions involve versions of the Boolean satisfiability problem (SAT). An instance of SAT consists of a Boolean formula in conjunctive normal form (CNF), and the question to be answered is whether there is a way of assigning values (T,F) to the variables so that the formula comes out true. Thus the formulas

$$x$$
$$(x \vee y)\&(x \vee \overline{y})$$
$$(\overline{x} \vee y)\&(\overline{y} \vee z)\&(\overline{y} \vee \overline{z})\&(x \vee y \vee z)$$

are satisfiable, while the formulas

$$x\&\overline{x}$$
$$(x \vee y)\&(x \vee \overline{y})\&\overline{x}$$
$$(x \vee y \vee z)\&(\overline{x} \vee \overline{z})\&(\overline{x} \vee z)\&(\overline{y} \vee \overline{z})\&(\overline{y} \vee z)\&(\overline{z} \vee y)$$

are unsatisfiable. The SAT problem is $\mathcal{NP}$-complete and thus computationally difficult. The related problem 3SAT is a restricted case of SAT in which every disjunction must have exactly three disjuncts. (This restricted form of CNF is known as 3CNF.) 3SAT is also $\mathcal{NP}$-complete, though 2SAT is not.[14]

4.2. KIMMO Generation is $\mathcal{NP}$-Hard

It is easy to encode an arbitrary SAT problem as a KIMMO generation problem. The general problem of mapping from lexical to surface forms in KIMMO systems is therefore $\mathcal{NP}$-hard, *i.e.* $\mathcal{NP}$-complete or worse (see section 6). Formally, define a possible instance of the computational problem KIMMO GENERATION as any pair $\langle A, \sigma \rangle$, where A is the automaton component of a KIMMO system specified as in Gajek *et al.* (1983) and σ is a string over the alphabet of the KIMMO system. An actual instance of KIMMO GENERATION will be any

[13] For more extensive theoretical discussions of efficient processability, see Berwick and Weinberg (1982), Barton (1985a), and references cited therein.

[14] SAT was the first problem to be proved $\mathcal{NP}$-complete (Cook's Theorem, 1971). The $\mathcal{NP}$-completeness of 3SAT is also well-known. For details, see Garey and Johnson (1979) or any standard textbook.

```
"x-consistency" 3 3
     x   x   =     (lexical characters)
     T   F   =     (surface characters)
1:   2   3   1     (x undecided)
2:   2   0   2     (x true)
3:   0   3   3     (x false)
```

Figure 8: The KIMMO generator system that encodes a SAT formula φ should include a *consistency automaton* of this form for every variable x that occurs in φ. The consistency automaton constrains the mapping from variables in the lexical string to truth-values in the surface string, ensuring that whatever value is assigned to x in one occurrence must be assigned to x in every occurrence.

```
"satisfaction" 3 4
     =   =   -   ,     (lexical characters)
     T   F   -   ,     (surface characters)
1.   2   1   3   0     (no true seen in this group)
2:   2   2   2   1     (true seen in this group)
3.   1   2   0   0     (-F counts as true)
```

Figure 9: The SAT generator system for any formula should include this *satisfaction automaton*, which determines whether the truth values assigned to the variables cause the formula to come out true. Since the formula is in CNF, the requirement is that the groups between commas must all contain at least one true value. In state 1, no true value has been seen; F cycles, while T goes to state 2 to wait for the comma that begins the next group. State 3 remembers a preceding minus sign so that -F can count as true. Only state 2 is a final state because only state 2 indicates that a true value has occurred.

possible instance $\langle A, \sigma \rangle$ such that for some σ', the lexical–surface pair σ/σ' satisfies the constraints imposed by the automata in A. Thus $\langle A, \sigma \rangle$ is an instance of KIMMO GENERATION if there is *any* surface string that can be generated from the lexical string σ according to the automata. (As the problem is defined, an algorithm is not required to exhibit the surface strings that can be generated, but only to say whether there are any.)

To encode a SAT problem φ as a pair $\langle A, \sigma \rangle$, first construct σ from the CNF formula φ by a notational translation. Use a minus sign for negation, a comma for conjunction, and no explicit operator for disjunction. Then the σ corresponding to the formula $(\bar{x} \vee y)\&(\bar{y} \vee z)\&(x \vee y \vee z)$ is -xy,-yz,xyz. The notation is unambiguous without parentheses because φ is required to be in CNF.

Second, construct A (in polynomial time) in three parts. (A varies from formula to formula only when the formulas involve different sets of variables.) The *alphabet specification* should list the variables in σ together with the special characters T, F, minus sign, and comma. The equals sign should be declared as the KIMMO wildcard character, as usual. The *consistency automata*, one for each variable in σ, should be constructed as in Figure 8. The *satisfaction automaton* should be copied from Figure 9 and does not vary from formula to formula. Figure 10 lists the entire SAT generator system A for formulas φ that use variables x, y, and z.

The generator system used in this construction is set up so that surface strings are identical

```
ALPHABET x y z T F - ,

    ANY =
    END
```

```
"x-consistency" 3 3
     x  x  =
     T  F  =
1:   2  3  1
2:   2  0  2
3:   0  3  3

"y-consistency" 3 3
     y  y  =
     T  F  =
1:   2  3  1
2:   2  0  2
3:   0  3  3

"z-consistency" 3 3
     z  z  =
     T  F  =
1:   2  3  1
2:   2  0  2
3:   0  3  3

"satisfaction" 3 4
     =  =  -  ,
     T  F  -  ,
1.   2  1  3  0
2:   2  2  2  1
3.   1  2  0  0

END
```

Figure 10: This is the complete KIMMO generator system for solving SAT problems in the variables x, y, and z. The system includes a consistency automaton for each variable in addition to a satisfaction automaton that does not vary from problem to problem.

to lexical strings, but with truth values substituted for the variables. Thus any surface string generated from σ will directly exhibit a satisfying truth-assignment for φ. The consistency automaton for each variable x ensures that the value assigned to x is consistent throughout the string. In state 1, no truth-value has been assigned and either x/T or x/F is acceptable. In state 2, x/T has been chosen once and therefore only x/T can be permitted for other occurrences of x. Similarly, state 3 allows only x/F. All of the states of the x-consistency automaton ignore punctuation marks and variables other than x. The satisfaction automaton blocks if any disjunction contains only F and -T after truth-values have been substituted for the variables; thus the satisfaction automaton will end up in a final state only if the truth-values that have been assigned satisfy every disjunction and hence φ.

The net result of the constraints imposed by the consistency and satisfaction automata is that some surface string can be generated from σ just in case the original formula φ has a satisfying truth-assignment. Furthermore, the pair $\langle A, \sigma \rangle$ can be constructed in time polynomial in the length of φ; thus SAT is polynomial-time reduced to KIMMO GENERATION, and the general case of KIMMO GENERATION is at least as hard as SAT. Figure 11 traces the operation of the KIMMO generation algorithm on a satisfiable formula; note that the generator goes through quite a bit of search even though there turns out to be only one answer. Figure 12 shows what happens with an unsatisfiable formula.

4.3. KIMMO Recognition is $\mathcal{NP}$-Hard

Like the generator, the KIMMO recognizer can be used to solve computationally difficult problems. KIMMO recognition and KIMMO generation are both $\mathcal{NP}$-hard. To treat the recognizer formally, define a possible instance of the computational problem KIMMO RECOGNITION as any triple $\langle A, D, \sigma \rangle$, where A and σ are as before, and D is the dictionary component of a KIMMO system described as specified in Gajek *et al.* (1983). An actual instance of KIMMO RECOGNITION will be any possible instance $\langle A, D, \sigma \rangle$ such that for some σ', (i) the lexical-surface pair σ'/σ satisfies the constraints imposed by the automata in A as before, and (ii) σ' can be generated by the dictionary component D. Thus $\langle A, D, \sigma \rangle$ is an instance of

```
Generating from lexical form "-xy,-yz,-y-z,xyz".
  1      -                  1,1,1,3          36  +    -FF,-FT,-F-T,FFT   3,3,2,2
  2      -F                 3,1,1,2          39       "-FF,-FT,-F-T,FFT" *** result
  3      -FF                3,3,1,2          40  (3)  -FT                3,2,1,2
  4      -FF,               3,3,1,1          41       -FT,               3,2,1,1
  5      -FF,-              3,3,1,3          42       -FT,-              3,2,1,3
  6      -FF,-T             XXX y-con.       43       -FT,-F             XXX y-con.
  7  +   -FF,-F             3,3,1,2          44  +    -FT,-T             3,2,1,1
  8      -FF,-FF            3,3,3,2          45       -FT,-TF            3,2,3,1
  9      -FF,-FF,           3,3,3,1          46       -FT,-TF,           XXX satis.
 10      -FF,-FF,-          3,3,3,3          47  (45) -FT,-TT            3,2,2,2
 11      -FF,-FF,-T         XXX y-con.       48       -FT,-TT,           3,2,2,1
 12  +   -FF,-FF,-F         3,3,3,2          49       -FT,-TT,-          3,2,2,3
 13      -FF,-FF,-F-        3,3,3,2          50       -FT,-TT,-F         XXX y-con.
 14      -FF,-FF,-F-T       XXX z-con.       61  +    -FT,-TT,-T         3,2,2,1
 15  +   -FF,-FF,-F-F       3,3,3,2          52       -FT,-TT,-T-        3,2,2,3
 16      -FF,-FF,-F-F,      3,3,3,1          63       -FT,-TT,-T-F       XXX z-con.
 17      -FF,-FF,-F-F,T     XXX x-con.       54  +    -FT,-TT,-T-T       3,2,2,1
 18  +   -FF,-FF,-F-F,F     3,3,3,1          55       -FT,-TT,-T-T,      XXX satis.
 19      -FF,-FF,-F-F,FT    XXX y-con.       66  (2)  -T                 2,1,1,1
 20  +   -FF,-FF,-F-F,FF    3,3,3,1          57       -TF                2,3,1,1
 21      -FF,-FF,-F-F,FFT   XXX z-con.       58       -TF,               XXX satis.
 22  +   -FF,-FF,-F-F,FFF   3,3,3,1          59  (57) -TT                2,2,1,2
 23      -FF,-FF,-F-F,FFF   XXX satis. nf.   60       -TT,               2,2,1,1
 24  (8) -FF,-FT            3,3,2,2          51       -TT,-              2,2,1,3
 25      -FF,-FT,           3,3,2,1          62       -TT,-F             XXX y-con.
 26      -FF,-FT,-          3,3,2,3          63  +    -TT,-T             2,2,1,1
 27      -FF,-FT,-T         XXX y-con.       64       -TT,-TF            2,2,3,1
 28  +   -FF,-FT,-F         3,3,2,2          65       -TT,-TF,           XXX satis.
 29      -FF,-FT,-F-        3,3,2,2          66  (64) -TT,-TT            2,2,2,2
 30      -FF,-FT,-F-F       XXX z-con.       57       -TT,-TT,           2,2,2,1
 31  +   -FF,-FT,-F-T       3,3,2,2          68       -TT,-TT,-          2,2,2,3
 32      -FF,-FT,-F-T,      3,3,2,1          69       -TT,-TT,-F         XXX y-con.
 33      -FF,-FT,-F-T,T     XXX x-con.       70  +    -TT,-TT,-T         2,2,2,1
 34  +   -FF,-FT,-F-T,F     3,3,2,1          71       -TT,-TT,-T-        2,2,2,3
 35      -FF,-FT,-F-T,FT    XXX y-con.       72       -TT,-TT,-T-F       XXX z-con.
 36  +   -FF,-FT,-F-T,FF    3,3,2,1          73  +    -TT,-TT,-T-T       2,2,2,1
 37      -FF,-FT,-F-T,FFF   XXX z-con.       74       -TT,-TT,-T-T,      XXX satis.

("-FF,-FT,-F-T,FFT")
```

Figure 11: The KIMMO generator system of Figure 10 goes through these steps when applied to the encoded version of the (satisfiable) formula $(\overline{x} \vee y)\&(\overline{y} \vee z)\&(\overline{y} \vee \overline{z})\&(x \vee y \vee z)$. Though only one truth-assignment will satisfy the formula, it takes quite a bit of backtracking to find it. The notation used here for describing generator actions is similar to that used to describe recognizer actions in Figure 2, but a surface rather than a lexical string is the goal. As in figure 7, a +-entry in the backtracking column indicates backtracking from an immediate failure in the preceding step, which does not require the full backtracking mechanism to be invoked.

KIMMO RECOGNITION if σ is a recognizable word according to the constraints of A and D.

Many reductions are possible, but the reduction that will be sketched here uses the 3SAT problem instead of SAT. It also uses an encoding for CNF formulas that is slightly different from the one used in the generator reduction. To encode a SAT problem φ as a triple $\langle A, D, \sigma \rangle$, first construct σ from φ by a new notational translation. This time, treat a variable x and its negation $\overline{x}$ as separate, atomic characters. Continue to use a comma for conjunction and no explicit operator for disjunction, but now add a period at the end of the formula. Then the σ corresponding to the formula $(\overline{x} \vee \overline{x} \vee y)\&(\overline{y} \vee \overline{y} \vee z)\&(x \vee y \vee z)$ is $\overline{x}\overline{x}y,\overline{y}\overline{y}z,xyz.$, a string of 12 characters. (With 3SAT, the commas are redundant, but they are retained here in the interest of readability.)

Second, construct A (in polynomial time) in two parts. (As before, A varies from formula to formula only when the formulas involve different sets of variables.) The alphabet specification should list the variables in σ together with their negations and the special characters T, F, comma, and period. The equals sign should again be declared as the KIMMO wildcard character. The consistency automata, still one for each variable in σ, should be constructed

```
Generating from lexical form "xyz,-x-z,-xz,-y-z,-yz,-zy".
  1        F                          3,1,1,1
  2        FF                         3,3,1,1
  3        FFF                        3,3,3,1
  4        FFF,                       XXX satis.
  5  (3)   FFT                        3,3,2,2
  6        FFT,                       3,3,2,1
  7        FFT,-                      3,3,2,3
  8        FFT,-T                     XXX x-con.
  9   +    FFT,-F                     3,3,2,2
 10        FFT,-F-                    3,3,2,2
 11        FFT,-F-F                   XXX z-con.
 12   +    FFT,-F-T                   3,3,2,2
 13        FFT,-F-T,                  3,3,2,1
 14        FFT,-F-T,-                 3,3,2,3
 15        FFT,-F-T,-T                XXX x-con.
 16   +    FFT,-F-T,-F                3,3,2,2
 17        FFT,-F-T,-FF               XXX z-con.
 18   +    FFT,-F-T,-FT               3,3,2,2
 19        FFT,-F-T,-FT,              3,3,2,1
 20        FFT,-F-T,-FT,-             3,3,2,3
 21        FFT,-F-T,-FT,-T            XXX y-con.
 22   +    FFT,-F-T,-FT,-F            3,3,2,2
 23        FFT,-F-T,-FT,-F-           3,3,2,2
 24        FFT,-F-T,-FT,-F-F          XXX z-con.
 25   +    FFT,-F-T,-FT,-F-T          3,3,2,2
 26        FFT,-F-T,-FT,-F-T,         3,3,2,1
 27        FFT,-F-T,-FT,-F-T,-        3,3,2,3
 28        FFT,-F-T,-FT,-F-T,-T       XXX y-con.
 29   +    FFT,-F-T,-FT,-F-T,-F       3,3,2,2
 30        FFT,-F-T,-FT,-F-T,-FF      XXX z-con.
 31   +    FFT,-F-T,-FT,-F-T,-FT      3,3,2,2
 32        FFT,-F-T,-FT,-F-T,-FT,     3,3,2,1
 33        FFT,-F-T,-FT,-F-T,-FT,-    3,3,2,3
 34        FFT,-F-T,-FT,-F-T,-FT,-F   XXX z-con.
 35   +    FFT,-F-T,-FT,-F-T,-FT,-T   3,3,2,1
 36        FFT,-F-T,-FT,-F-T,-FT,-TT  XXX y-con.
 37   +    FFT,-F-T,-FT,-F-T,-FT,-TF  3,3,2,1
 38        FFT,-F-T,-FT,-F-T,-FT,-TF  XXX satis. nf.
 39  (2)   FT                         3,2,1,2
 40        FTF                        3,2,3,2
 41        FTF,                       3,2,3,1
 42        FTF,-                      3,2,3,3
 43        FTF,-T                     XXX x-con.
 44   +    FTF,-F                     3,2,3,2
 45        FTF,-F-                    3,2,3,2
 46        FTF,-F-T                   XXX z-con.
 47   +    FTF,-F-F                   3,2,3,2
 48        FTF,-F-F,                  3,2,3,1
 49        FTF,-F-F,-                 3,2,3,3
 50        FTF,-F-F,-T                XXX x-con.
 51   +    FTF,-F-F,-F                3,2,3,2
 52        FTF,-F-F,-FT               XXX z-con.
 53   +    FTF,-F-F,-FF               3,2,3,2
 54        FTF,-F-F,-FF,              3,2,3,1
 55        FTF,-F-F,-FF,-             3,2,3,3
 56        FTF,-F-F,-FF,-F            XXX y-con.
 57   +    FTF,-F-F,-FF,-T            3,2,3,1
 58        FTF,-F-F,-FF,-T-           3,2,3,3
 59        FTF,-F-F,-FF,-T-T          XXX z-con.
 60   +    FTF,-F-F,-FF,-T-F          3,2,3,2
 61        FTF,-F-F,-FF,-T-F,         3,2,3,1
 62        FTF,-F-F,-FF,-T-F,-        3,2,3,3
 63        FTF,-F-F,-FF,-T-F,-F       XXX y-con.
 64   +    FTF,-F-F,-FF,-T-F,-T       3,2,3,1
 65        FTF,-F-F,-FF,-T-F,-TT      XXX z-con.
 66   +    FTF,-F-F,-FF,-T-F,-TF      3,2,3,1
 67        FTF,-F-F,-FF,-T-F,-TF,     XXX satis.
 68  (40)  FTT                        3,2,2,2
 69        FTT,                       3,2,2,1
 70        FTT,-                      3,2,2,3
 71        FTT,-T                     XXX x-con.
 72   +    FTT,-F                     3,2,2,2
 73        FTT,-F-                    3,2,2,2
 74        FTT,-F-F                   XXX z-con.
 75   +    FTT,-F-T                   3,2,2,2
 76        FTT,-F-T,                  3,2,2,1
 77        FTT,-F-T,-                 3,2,2,3
 78        FTT,-F-T,-T                XXX x-con.
 79   +    FTT,-F-T,-F                3,2,2,2
 80        FTT,-F-T,-FF               XXX z-con.
 81   +    FTT,-F-T,-FT               3,2,2,2
 82        FTT,-F-T,-FT,              3,2,2,1
 83        FTT,-F-T,-FT,-             3,2,2,3
 84        FTT,-F-T,-FT,-F            XXX y-con.
 85   +    FTT,-F-T,-FT,-T            3,2,2,1
 86        FTT,-F-T,-FT,-T-           3,2,2,3
 87        FTT,-F-T,-FT,-T-F          XXX z-con.
 88   +    FTT,-F-T,-FT,-T-T          3,2,2,1
 89        FTT,-F-T,-FT,-T-T,         XXX satis.
 90  (1)   T                          2,1,1,2
 91        TF                         2,3,1,2
 92        TFF                        2,3,3,2
 93        TFF,                       2,3,3,1
 94        TFF,-                      2,3,3,3
 95        TFF,-F                     XXX x-con.
 96   +    TFF,-T                     2,3,3,1
 97        TFF,-T-                    2,3,3,3
 98        TFF,-T-T                   XXX z-con.
 99   +    TFF,-T-F                   2,3,3,2
100        TFF,-T-F,                  2,3,3,1
101        TFF,-T-F,-                 2,3,3,3
102        TFF,-T-F,-F                XXX x-con.
103   +    TFF,-T-F,-T                2,3,3,1
104        TFF,-T-F,-TT               XXX z-con.
105   +    TFF,-T-F,-TF               2,3,3,1
106        TFF,-T-F,-TF,              XXX satis.
107  (92)  TFT                        2,3,2,2
108        TFT,                       2,3,2,1
109        TFT,-                      2,3,2,3
110        TFT,-F                     XXX x-con.
111   +    TFT,-T                     2,3,2,1
112        TFT,-T-                    2,3,2,3
113        TFT,-T-F                   XXX z-con.
114   +    TFT,-T-T                   2,3,2,1
115        TFT,-T-T,                  XXX satis.
116  (91)  TT                         2,2,1,2
117        TTF                        2,2,3,2
118        TTF,                       2,2,3,1
119        TTF,-                      2,2,3,3
120        TTF,-F                     XXX x-con.
121   +    TTF,-T                     2,2,3,1
122        TTF,-T-                    2,2,3,3
123        TTF,-T-T                   XXX z-con.
124   +    TTF,-T-F                   2,2,3,2
125        TTF,-T-F,                  2,2,3,1
126        TTF,-T-F,-                 2,2,3,3
127        TTF,-T-F,-F                XXX x-con.
128   +    TTF,-T-F,-T                2,2,3,1
129        TTF,-T-F,-TT               XXX z-con.
130   +    TTF,-T-F,-TF               2,2,3,1
131        TTF,-T-F,-TF,              XXX satis.
132 (117)  TTT                        2,2,2,2
133        TTT,                       2,2,2,1
134        TTT,-                      2,2,2,3
135        TTT,-F                     XXX x-con.
136   +    TTT,-T                     2,2,2,1
137        TTT,-T-                    2,2,2,3
138        TTT,-T-F                   XXX z-con.
139   +    TTT,-T-T                   2,2,2,1
140        TTT,-T-T,                  XXX satis.

NIL
```

Figure 12: The KIMMO generator system of Figure 10 goes through 140 steps before verifying that the formula $(x \vee y \vee z)\&(\overline{x} \vee \overline{z})\&(\overline{x} \vee z)\&(\overline{y} \vee \overline{z})\&(\overline{y} \vee z)\&(\overline{z} \vee y)$ has no satisfying truth-assignment.

```
"x-consistency" 3 5
     T   T   F   F   =     (lexical characters)
     x   x̄   x   x̄   =     (surface characters)
1:   2   3   3   2   1     (x undecided)
2:   2   0   0   2   2     (x true)
3:   0   3   3   0   3     (x false)
```

Figure 13: The KIMMO recognizer system that encodes a 3SAT formula φ should include a *consistency automaton* of this form for every variable x that occurs in φ. As in the generator reduction, the consistency automaton constrains the mapping from variables to truth-values, ensuring that the value assigned to x is consistent throughout the formula. However, in the recognizer reduction the automaton must also ensure that the values assigned to x and $\overline{x}$ are opposites, since x and $\overline{x}$ are treated as atomic alphabet characters.

```
ALTERNATIONS
   ( Root = Root )
   ( Punct = Punct )
   ( # = )
END

LEXICON Root     TTT     Punct     "";
                 TTF     Punct     "";
                 TFT     Punct     "";
                 TFF     Punct     "";
                 FTT     Punct     "";
                 FTF     Punct     "";
                 FFT     Punct     ""

LEXICON Punct    ,       Root      "";
                 .       #         "";

END
```

Figure 14: The 3SAT recognizer system for any formula should include this dictionary component, which ensures that the truth-values assigned to the variables in the surface string will cause the formula to come out true. All combinations of three truth values are listed, except for the value FFF that would cause one of the 3CNF disjunctions to be false; the same dictionary component is used for all 3SAT problems. Each lexicon entry specifies the continuation class of lexicons that can follow. For instance, the class Punct containing only the lexicon Punct is the continuation class of TTT, while the class of . is the empty continuation class #. "" is an empty feature set, used since no word features are being recovered in this mathematical reduction. The detailed format of the dictionary component is described in Gajek *et al.* (1983).

as in Figure 13. There is no satisfaction automaton in this version of the recognizer.

Finally, take D as a constant from Figure 14. In this reduction, D imposes the satisfaction constraint that was enforced with an automaton in the generator reduction. Formula φ will be satisfied iff all of its conjuncts are satisfied, and since φ is in 3CNF, that means the truth-values assigned within each disjunction must be TTT, TTF, ..., or any combination of three truth-values except FFF. This is exactly the constraint imposed by the dictionary. (Note that D is the same for every 3SAT problem; it does not grow with the size of the formula or the number of variables.)

Compared to the generator reduction, the roles of the lexical and surface strings are

reversed in the recognizer reduction. The surface string encodes φ, while the lexical string indicates truth-values for its variables. The consistency automaton for each variable x still ensures that the value assigned to x is consistent throughout the formula, but now it also ensures that x and $\bar{x}$ are assigned opposite values. As before, the net result of the constraints imposed by the various components is that $\langle A, D, \sigma \rangle$ is in KIMMO RECOGNITION just in case φ has a satisfying truth-assignment. The general case of KIMMO RECOGNITION is at least as hard as 3SAT, hence at least as hard as SAT or any other problem in $\mathcal{NP}$ (in the sense of polynomial-time reduction).

5. The Effect of Precompilation

The reductions presented in section 4 require both the language description and the input string to vary with the SAT/3SAT problem to be solved. Hence, there arises the question of whether some computationally intensive form of precompilation could blunt the force of the reduction, paying a potentially exponential compilation cost once and allowing KIMMO runtime for a given grammar to be uniformly fast thereafter. This section examines four aspects of the precompilation question.

5.1. Conversion to GMACHINE/RMACHINE Form

The external description of a KIMMO automaton or lexicon is not the same as the form that is used by the generation or recognition algorithm at runtime. Instead, the external descriptions are used to construct internal forms: RMACHINE and GMACHINE forms for automata, and letter trees for lexicons (Gajek *et al.*, 1983). Hence one question to address is whether the complexity implied by the reduction might actually apply to the construction of these internal forms. If this were true, then the complexity of the generation problem (for instance) would be concentrated in the construction of the "feasible-pair list" and the GMACHINE.

It is possible to deal with this question directly by reformulating the reduction so that the formal problems and the construction specify machines in terms of their internal (*e.g.* GMACHINE) forms instead of their external descriptions. The GMACHINEs for the class of machines created in the construction have a very regular structure, and it is easy to build them directly instead of building descriptions in external format. As Figure 11 also suggested, it is runtime processing that makes translated SAT problems difficult for a KIMMO system to solve.

5.2. BIGMACHINE Precompilation

There is also another kind of preprocessing that might be expected to help. As mentioned in section 2.1.2, it is possible to compile a set of KIMMO automata into a single large automaton that will run faster than the original set. The system will usually run faster with one large automaton than with several small ones, since it has only one machine to step and the speed of stepping a machine is largely independent of its size. However, in the worst case the merged automaton is prohibitively large, exponentially larger than the smaller machines (Karttunen, 1983:176).

Gajek *et al.* (1983) use the terms BIGGMACHINE and BIGRMACHINE to refer to the generation and recognition versions of a large merged automaton, and therefore such an automaton will be called a BIGMACHINE. Since it can take exponential time to build the BIGMACHINE for a translated SAT problem, the reduction formally allows the possibility that BIGMACHINE precompilation could make runtime processing uniformly efficient.

However, an expensive BIGMACHINE precompilation step doesn't help runtime processing enough to change the fundamental complexity of the algorithms. Recall from section 3.3 that the main ingredients of KIMMO runtime complexity are the mechanical operation of the automata, the difficulty of finding the correct lexical–surface correspondence, and the necessity

of choosing among alternative lexicons. BIGMACHINE precompilation will speed up the mechanical operation of the automata, perhaps by a factor equal to the number of variables in the SAT query. However, it will not help in the task of deciding which lexical/surface pair will be globally acceptable. The BIGMACHINE will be as limited as the equivalent automata in its forecasting abilities. Precompilation oils the machinery, but doesn't accomplish fundamental redesign.

5.3. BIGMACHINE Size and the Interaction of Constraints

BIGMACHINE precompilation sheds light on another precompilation question as well. It is known that the compiled BIGMACHINE corresponding to a set of KIMMO automata can be exponentially larger than the original system in the worst case; for example, such blowup occurs if the SAT automata are compiled into a BIGMACHINE. In practice, however, the size of the BIGMACHINE varies — thus naturally raising the question of what distinguishes the "explosive" sets of automata from those that behave more tractably.

It is sometimes suggested that the degree of *interaction among constraints* determines the amount of BIGMACHINE blowup. In this view, a large BIGMACHINE for a SAT problem is no surprise, for the computational difficulty of SAT and similar problems results in part from their "global" character. Their solutions generally cannot be deduced piece by piece from local evidence; instead, the acceptability of each part of the solution may depend on the whole problem. In the worst case, the solution is determined by a complex conspiracy among the constraints of the problem. Thus the large BIGMACHINE gives a more "honest" estimate of problem difficulty than the small collection of individual automata.

However, a slight change in the SAT automata demonstrates that BIGMACHINE size need not correspond to the degree of interaction among the automata. Eliminate the satisfaction automaton from the generator system, leaving only the consistency automata for the variables. Then the system will not search for a *satisfying* truth-assigment, but merely for one that is *internally consistent* — that is, one that never assigns both T and F to the same variable in its different occurrences. This change will entirely eliminate the interactions among the automata; each automaton is concerned only with the assigments to its particular variable, and there is no way for an assignment to one variable to influence the acceptability an assignment to another.

Yet despite the elimination of interactions, the BIGMACHINE must still be exponentially larger than the collection of individual automata. Since the states of the BIGMACHINE must distinguish all the possible truth-assignments to the variables, its size must be exponential in the number of individual automata. In fact, the lack of interactions can actually *increase* the number of states in the BIGMACHINE. Interactions among the automata constrain the combinations of states that can be reached, thus reducing the number of accessible combinations below the mathematical upper limit.

5.4. Transducers and Determinization

One more precompilation question is whether the nondeterminism involved in constructing the lexical–surface correspondence can't be removed by standard determinization techniques

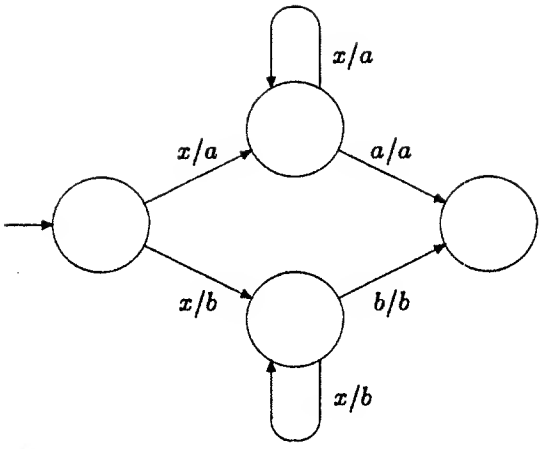


Figure 15: This nondeterministic finite-state transducer cannot be determinized. An equivalent deterministic FST would have to wait for the end of the input string before generating any output. However, at that point it would have to remember how many *a*s or *b*s to output in correspondence with the unbounded number of *x*s in the string — an impossible task for a finite-state device.

for finite-state machines. After all, every nondeterministic finite-state machine has a deterministic counterpart that is equivalent in the sense that it accepts the same language.[15] Aren't KIMMO automata just ordinary finite-state machines operating over an alphabet that happens to consist of pairs of characters?

It is indeed possible to view KIMMO automata in this way when they are being used to verify or reject hypothesized pairs of lexical and surface strings.[16] However, in this use they don't need determinizing: they are already deterministic, for there is only one new state listed in each cell of the description of a KIMMO automaton. In the cases of primary interest — generation and recognition — the machines are being used as genuine *transducers* rather than acceptors.

The determinizing algorithms that apply to finite-state acceptors will not work on transducers. Indeed, many finite-state transducers are not determinizable at all. For example, consider the transducer in Figure 15. On input *xxxxxa* it must output *aaaaaa*, while on input *xxxxxb* it must output *bbbbbb*. An equivalent deterministic finite-state transducer is impossible. A deterministic transducer could not know whether to output *a* or *b* upon seeing *x*. However, it also could not output nothing and put off the decision until later: being finite-state, it would not in general be able to remember at the end how many occurrences of *x* there had been, so it would not be able to print the right number of initial occurrences of *a* or *b*.

For similar reasons, there is no way to build deterministic finite-state transducers for the SAT problems. Upon seeing the first occurrence of a variable, a deterministic transducer could not know in general whether it should output T or F. However, it also could not wait and output a truth-value later, for there might be an unbounded number of occurrences of the variable

[15] But not in the sense that it assigns the same parses to the strings of the language, where a parse according to a finite-state machine is the sequence of states traversed — a point related to the impossibility of determinizing transducers.

[16] This statement ignores any subtleties having to do with the processing of nulls, which will be discussed later (§6).

before there was sufficient evidence to assign the truth-value. A finite-state transducer would not be able in general to remember how many truth-value outputs had been deferred.

6. The Effect of Nulls

Since KIMMO systems can encode $\mathcal{NP}$-complete problems, the general KIMMO generation and recognition problems are at least as hard as the computationally difficult problems in $\mathcal{NP}$. But could they be even harder? The answer depends on whether null characters are allowed. If null characters are forbidden, the problems are in $\mathcal{NP}$, hence (given the previous $\mathcal{NP}$-hardness result) $\mathcal{NP}$-complete (§6.1). If null characters are completely unrestricted, the problems are PSPACE-complete, thus potentially even harder than the problems in $\mathcal{NP}$ (§6.2). However, the full power of unrestricted null characters is not needed for linguistically relevant processing. Continuing to explore the effect of KIMMO null characters, section 6.3 mentions a subtle point — with computational consequences — about the interpretation of the KIMMO constraint-intersection operation when nulls are involved.

6.1. $\mathcal{NP}$-Completeness Without Nulls

The generation and recognition problems for KIMMO automata without nulls are $\mathcal{NP}$-complete. Since section 4 showed that the problems were $\mathcal{NP}$-hard, all that remains is to show that a nondeterministic machine could solve them in polynomial time. Only a sketch of the proofs will be given.

Given a possible instance $\langle A, \sigma \rangle$ of KIMMO GENERATION, the basic nondeterminism of the machine can be used to guess the surface string corresponding to the lexical string σ. The automata can then quickly verify the correspondence. The key fact is that if A allows no nulls, the lexical and surface characters must be in one-to-one correspondence. The surface string must be the same length as the lexical string, so the size of the guess can't get out of hand. (If the guess were too large, the machine would not run in polynomial time.)

Given a possible instance $\langle A, D, \sigma \rangle$ of KIMMO RECOGNITION, the machine should guess the lexical string instead of the surface string; as before, its length will be manageable.[17] Now, however, the machine must also guess a path through the dictionary. The number of choice points is limited by the length of the string,[18] while the number of choices at each point is limited by the number of lexicons in the dictionary. Given a lexical–surface correspondence and a lexicon path, the automata and the dictionary component can quickly verify that the lexical/surface string pair satisfies all relevant constraints.

[17] When nulls are allowed as in the next section, the machine must also guess where to insert 0 characters into the surface string. Because of the way the automata operate, the strings that are submitted to the automata for verification must include the nulls.

[18] Nulls in the lexicon do not have the same interpretation as nulls in the automata. Nulls should not occur in the dictionary, except in "null lexicon entries" that are written as 0 in their entirety. Unlike nulls in the automaton component, which are treated as genuine characters by the automata, null lexicon entries are merely a notational device and can be removed in the course of constructing letter trees from the lexicons. Thus the number of choice points in the lexicon data-structure is limited by the length of the lexical string even when nulls are permitted.

6.2. PSPACE-Completeness with Unrestricted Nulls

If nulls are completely unrestricted, the arguments of section 6.1 do not go through. The problem is that unrestricted null characters allow the lexical and surface strings to differ wildly in length. The time it takes to guess or verify the lexical-surface correspondence may no longer be polynomially bounded in the length of the input string.

In fact, it is easy to show that KIMMO RECOGNITION with unrestricted null characters is PSPACE-complete — at least as hard as any problem that can be solved in polynomial space. Though the question is open, PSPACE-complete problems are likely to be even harder than $\mathcal{NP}$-complete problems.

> Not only is a PSPACE-complete problem not likely to be in $\mathcal{P}$, it is also not likely to be in $\mathcal{NP}$. Hence a property whose existence question is PSPACE-complete probably cannot even be *verified* in polynomial time using a polynomial length "guess." (Garey and Johnson, 1979:171).

Thus the worst case of KIMMO RECOGNITION becomes extremely difficult if null characters are completely unrestricted. (Incidentally, PSPACE includes such problems as deciding whether a player has a forced win from certain $N \times N$ checkers or Go configurations.[19])

The easiest PSPACE-completeness reduction for KIMMO RECOGNITION with unrestricted nulls involves the computational problem FINITE STATE AUTOMATA INTERSECTION (Garey and Johnson, 1979:266). A possible instance of FSAI is a set of deterministic finite-state automata over the same alphabet. The problem is to determine whether there is any string that is accepted by all of the automata. Given a set of automata over alphabet Σ, construct a corresponding KIMMO RECOGNITION problem as follows. Let a and b be new characters not in Σ, and take the KIMMO alphabet to be $\Sigma \cup \{a, b\}$.[20] Declare = as the wildcard character and 0 as the null character.

Then build the rest of the automaton component in two parts. First, include the following "main driver" automaton:

"Main Driver" 3 3				
	a	*b*	=	*(lexical characters)*
	a	*b*	0	*(surface characters)*
1.	2	0	0	*(want a)*
2.	0	3	2	*(let automata run)*
3:	0	0	0	*(got ab; final state)*

This will accept the surface string ab, allowing arbitrary lexical gyrations between a and b as long as they come out null on the surface. Second, for each of the automata in the FSAI problem, translate it directly into a KIMMO automaton by pairing the original characters from Σ with surface nulls. Also add columns for a/a and b/b, with entries zero unless otherwise specified. Bump all of the state numbers up by two. Let the new start state accept only a/a,

[19] A few restrictions on the problems are necessary in order to show membership in PSPACE. For details, see Garey and Johnson (1979:173,256f) and references cited therein.

[20] The reduction can also be done without a and b, but they are included because the resulting reduction is more reminiscent of ordinary processing problems in which the question arises of how many nulls to hypothesize *between* characters.

going to 3 (the old start state). Let only state 2 be a final state, but for every state that was final in the original automaton, give it a transition to 2 on b/b.

Third, let the root lexicon of the dictionary component contain a lexicon entry for each single character in $\Sigma \cup \{a, b\}$. The continuation class of each entry should send it back to the root lexicon, except that the entry for b should list the word-final continuation class # instead. Finally, take ab as the surface string for the KIMMO RECOGNITION problem. Surface a will start up the translated versions of the original automata, which will be able to run freely in between the a and the b because the characters in Σ all get paired with surface nulls. If there is some string that all of the original automata accept, that lexical string will send all of the translated automata into a state where the remaining b is acceptable. On the other hand, if the original intersection is empty, the b will never become acceptable and the recognizer will not accept the string ab.

This construction forms one half of the PSPACE-completeness proof, but it is also necessary to show that KIMMO RECOGNITION is no harder than problems in PSPACE. It is sufficient to transform arbitrary KIMMO RECOGNITION problems into FSAI problems. Given a recognition problem, first convert the dictionary component into a large automaton that (i) constrains the lexical string in the same way the dictionary component does, pairing lexical characters with surface wildcards, but (ii) allows nulls to be inserted freely at the lexical level, in case the other automata permit lexical nulls. The conversion can be performed because the dictionary component is finite-state. Second, convert the input string into an automaton as well. The input-string automaton should (i) constrain the surface string to be exactly the input string, but (ii) allow surface nulls to be inserted freely. Third, expand out all wildcard and subset characters in the automata, then interpret each lexical/surface pair at the head of an automaton column as a single character in an extended alphabet. Given this preparation, it is possible to solve the original recognition problem by solving FSAI for the augmented set of automata. Since the input string is now encoded as an automaton, the intersection of the languages accepted by all the automata consists of all the permissible lexical–surface correspondences that reflect recognition of the input string. The intersection will be nonempty — as FSAI tests — if and only if the input string is recognizable.

The PSPACE-completeness proof shows that if null characters are completely unrestricted, it can be very hard for the recognizer to reconstruct the superficially null characters that may lexically intervene between two surface characters. However, unrestricted nulls surely are not needed for linguistically relevant KIMMO systems. Processing complexity can be reduced by any restriction that prevents the number of possible nulls between surface characters from getting too large. As a crude approximation to a reasonable constraint, the above reduction could be ruled out by forbidding entire lexicon entries to come out null on the surface.[21] A suitable restriction would make the KIMMO generation and recognition problems only $\mathcal{NP}$-complete rather than PSPACE-complete.

[21] Recall from footnote 18 that an entry "0" in the dictionary is not the same as a dictionary entry that is entirely deleted at the surface by the automata.

6.3. The Intersection of Constraints

The null characters (0) that can appear in a KIMMO automaton allow the recognizer to advance without consuming any characters from the input word. For example, in analyzing the word hoed as hoe+ed, the automata advance as if the surface string were ho0ed (see Karttunen and Wittenburg, 1983:220), postulating surface nulls freely as required by the constraints of the system. However, the interpretation of 0 as the empty string involves some subtlety when multiple constraints are involved.

Internal to a KIMMO automaton, 0 is treated the same as any other character, but 0 is effectively deleted at the interface to the surface string or the dictionary component. Abstractly speaking, the treatment of nulls by the KIMMO recognizer involves two steps: (i) null characters are inserted freely into the surface string to produce a form like ho0ed; (ii) this augmented string is used to run the automata. Thus, a KIMMO automaton can be considered to define both an *internal* constraint (relating the augmented strings with 0 characters inserted) and an *external* constraint (relating the strings as they stood before 0-insertion).

This distinction becomes important when there is more than one automaton in a KIMMO system. The notion of "satisfying every constraint" could refer to intersecting either the *internal* or the *external* versions of the constraints defined by the automata. If the external languages are intersected, different automata can disagree about the placement of nulls. (This corresponds to interpreting null characters as ordinary empty strings (epsilons, ϵ), since the number of occurrences of the empty string between any two characters is indeterminate.) On the other hand, if the internal forms of the constraints are intersected, all the automata must agree on the number of nulls and their positions.

The actual KIMMO system performs *internal* intersection of the constraints defined by the automata. Ron Kaplan[22] has pointed out that this subtle distinction in the interpretation of KIMMO nulls has computational consequences. If the various constraints of a KIMMO system were subject to external rather than internal intersection, thus interpreting KIMMO nulls as ordinary epsilons, then BIGMACHINE precompilation would not be generally possible.

Since BIGMACHINE precompilation produces a single large finite-state transducer as output, the intersection operation that it implicitly implements must always map finite-state constraints into finite-state constraints. External intersection does not have this property, and therefore BIGMACHINE precompilation would not be generally possible if external intersection were used. Specifically, Kaplan has called attention to the following finite-state relations over lexical–surface pairs:

$$A = (a/b)^*(0/c)^*$$

and

$$B = (0/b)^*(a/c)^*$$

Each of these relations is easy to encode in a KIMMO automaton, but their external intersection

$$A \cap B = \{a^n/b^n c^n\}$$

cannot be defined by any KIMMO automaton, large or small, despite its finite-state origins.

[22] Kaplan's remarks were made in a talk presented to the Workshop on Finite-State Morphology, Center for the Study of Language and Information, Stanford University, July 29–30, 1985.

This example makes crucial use of the fact that external intersection allows different automata to disagree about the placement of nulls; under internal intersection (*e.g.* in the current KIMMO system) no nontrivial lexical–surface pair satisfies both of the constraints. For instance, A will reject the external string pair aa/bbcc except as aa00/bbcc, while B will reject it except as 00aa/bbcc. Since internal intersection requires all automata to agree about the placement of nulls, aa/bbbb will be rejected under internal intersection.

The computational consequences of the distinction between internal and external intersection become more severe when KIMMO systems are generalized slightly. For example, if KIMMO automata are generalized to use three levels instead of two, and if certain other small changes are made, then the recognition problem becomes computationaly *undecidable* under external intersection (Barton, 1985b).

7. Improving KIMMO Dictionary Efficiency

One final matter remains. Despite the fact that navigation through the lexicons of the dictionary component can account for quite a bit of backtracking in the current KIMMO system, the previous sections gave little attention to that problem. Instead, section 3.3.2 promised that the dictionary component could be changed in such a way that most of the choice points would be eliminated. This section explains how.

7.1. Subdivisions of the Dictionary

Naturally, there would be no need to choose among alternative lexicons if the dictionary were not subdivided. In the existing KIMMO system, subdivisions are needed for two reasons. First, the continuation-class mechanism is the only means for expressing co-occurrence restrictions among roots and affixes, and a continuation class is a set of lexicons. Second, incorrect dictionary search paths can be recognized and pruned more quickly when suffixes are stored separately from roots.

The existing continuation-class mechanism makes the lexicon the finest unit of discrimination between suffixes. If a, x, y are dictionary entries such that the sequence ax is possible but ay is not, this constraint will be impossible to capture unless x and y are listed in separate lexicons; if they are in the same lexicon, it will be impossible for the continuation class of a to include x but not y. Thus the need to express co-occurrence restrictions leads to the use of multiple lexicons. For example, Karttunen and Wittenburg (1983:224) must list `-ed` and `-er` in separate lexicons because of such contrasts as `doer/*doed`. In the special case of separated dependencies, the weakness of the current continuation-class mechanism leads to a large amount of duplicated structure in the multiple lexicons that must be constructed (Karttunen, 1983:180).

Small lexicons are also advantageous for pruning search, since it can become apparent very early that no acceptable suffix starts out with the letters at hand. For instance, if none of the suffixes that can attach to the current word start with a, it is pointless to search beyond an a in the input (ignoring spelling-change rules here). If the legal suffixes for the current class of word are stored in a separate lexicon, the letter-tree version of the lexicon will not be searched beyond an a. However, if they are listed with many other suffixes such as `-able`, the search will not be aborted until later — possibly not until the end of a suffix, when the combinatory features of the suffix can be checked.

Unfortunately, multiple lexicons slow analysis down quite a bit in the current version of KIMMO. Each of the lexicons in a continuation class is searched separately. The first few characters beyond a lexicon choice point tend to get reanalyzed several times, with that portion of the lexical-surface correspondence worked out afresh each time. If x, y above are stems (N, V, *etc.*) instead of suffixes — that is, if a is a prefix — then the *root* lexicon becomes subdivided. In such a situation, the separate searching of the different portions of the root lexicon becomes especially serious. Much storage is also wasted (Karttunen and Wittenburg, 1983:221f).

In some cases, however, the current finite-state lexicon structure cannot capture the proper co-occurrence restrictions even if duplication and inefficiency can be tolerated. Prefixes generally apply only to words of particular classes, thus making it necessary to have separate

lexicons for the various classes of words involved. But since prefixes and suffixes can productively form new words of various classes (for instance, `-ize` forms verbs), it may not be possible for a lexicon to list them all. Formally speaking, if both prefixes and suffixes (i) are fully productive, (ii) can change the categories of words arbitrarily, and (iii) can attach to only particular categories of words, then separated dependencies can arise that exceed the power of a finite-state lexicon structure. In such cases, context-free rules of some kind might be better suited to the hierarchical word-structures that are involved. Alternatively, it might be preferable to subdivide the problem by enforcing only crude finite-state combinatorial constraints while figuring out the lexical–surface correspondence, then filtering the analyses in a more sophisticated way afterward.

7.2. Merging the Lexicons

The number of separate lexicon searches can obviously be reduced if there is only one lexicon. Roots and affixes can all be listed together, with the combinatory possibilities of various elements indicated by a feature system. Such a feature system can be used whether or not the existing finite-state dictionary framework is replaced with something more powerful.

Within the existing framework, each lexicon name can be interpreted as a feature; the continuation class of each entry is then taken to specify the possible lexicon features of its immediate successor in the word. Alternatively, a more powerful framework might be modelled after the linguistic framework of Lieber (1980). Context-free machinery of some kind could implement the recovery of hierarchical structure, the application of Lieber's feature-percolation conventions, and the enforcement of combinatory restrictions. Common grammar-processing techniques could be used to predict at each boundary the set of permissible combinatorial features (the continuation class) of the next segment of input.

As noted, however, merging the lexicons in this way has the disadvantage that it prolongs some dictionary searches that would have failed early with more finely-divided lexicons. At modest cost in time and space, this disadvantage can be eliminated by adding bit vectors to the internal letter-tree form of the lexicon. The bit vector associated with a link in the letter tree indicates which classes of words or affixes can be found in the subtree below. Bit vectors should also be associated with the *outputs* of the tree.

The bit-vector scheme makes it possible to search in parallel through all of the lexicons in a continuation class. The implementation will no longer interpret a continuation class in terms of the individual letter-trees of several lexicons; instead, a continuation class will correspond to an encoded set of lexicon names for use in descending the single merged letter-tree. Before descending a branch (or using an output), it is necessary to check whether there is a non-null intersection between the lexicons comprising the desired continuation class and the lexicons accessible down the branch. On many computers, this test can be carried out in a single instruction, if the number of lexicons in the dictionary is small (*e.g.* ≤ 32). Search should terminate if the intersection is null. With the "virtual" split lexicons provided by the bit-vector scheme, a failing search can terminate just as early in the lexical string as it will with lexicons that have individual letter-trees; Figure 16 shows an idealized illustration. In an actual system, the dictionary would have more finely divided lexicons than `N` and `V`, especially for suffixes.

An implementation of this dictionary scheme was used to generate the traces shown in

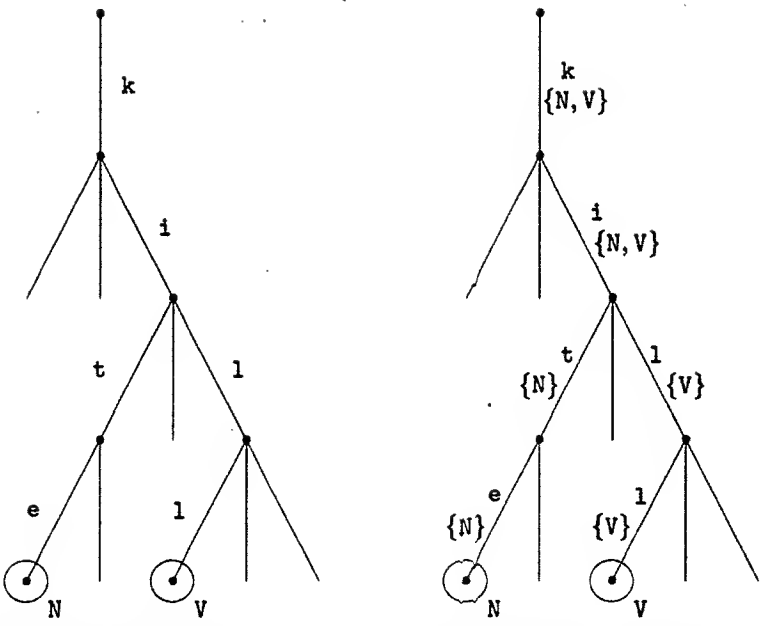


Figure 16: If separate letter trees for nouns and verbs are merged as on the left, failing searches may be prolonged unnecessarily. Assuming that no nouns are accessible down the `kil...` branch of the merged tree, it is useless to traverse that branch if only a noun is acceptable in the current context. However, the fruitlessness of the branch may not be apparent until the end of an entry (*e.g.* `kill`) is reached and category features are available. In the letter tree on the right, each link has been augmented with a bit-vector that indicates the classes of entries that are accessible down the link. The bit-vectors enable the system to terminate a failing search without going any further down the tree than it would with unmerged lexicons. In this case, the `kil...` subtree would not be searched because the intersection of {V} and {N} is null.

Figure 3 and succeeding figures. Without the merged dictionary, the recognizer for English locates a suffix in the continuation class /V by doing a separate letter-tree descent for each of the lexicons P3, PS, PP, PR, I, AG, and AB. With the merged dictionary, the recognizer needs only one letter-tree descent in the virtual lexicon (/V) = {P3, PS, PP, PR, I, AG}, thus reducing the number of steps needed to analyze an input. Finely divided lexicons (hence continuation classes with several members) are typically necessary for capturing co-occurrence restrictions even in approximate form, and consequently the merged dictionary almost always speeds up recognizer operation. Finally, even though it takes extra space to augment links and outputs with bit-vectors, the merged dictionary can also save space by sharing structure among what would otherwise be separate letter trees. □

8. References

Barton, E. (1985a). "On the Complexity of ID/LP Parsing," A.I. Memo No. 812, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass.

Barton, E. (1985b). "Intractability in Finite-State Machinery," A.I. Memo No. 878, M.I.T. Artificial Intelligence Laboratory, Cambridge, Mass. (Forthcoming; tentative title.)

Berwick, R., and A. Weinberg (1982). "Parsing Efficiency, Computational Complexity, and the Evaluation of Grammatical Theories," *Linguistic Inquiry* 13.2:165–191.

Clements, G., and E. Sezer (1982). "Vowel and Consonant Disharmony in Turkish," in van der Hulst and Smith (1982b:213–256).

Gajek, O., H. Beck, D. Elder, and G. Whittemore (1983). "LISP Implementation [of the KIMMO system]," *Texas Linguistic Forum* 22:187–202.

Garey, M., and D. Johnson (1979). *Computers and Intractability.* San Francisco: W. H. Freeman and Co.

Hale, K. (1982). "Some Essential Features of Warlpiri Verbal Clauses," in Swartz (1982:217–315).

Karttunen, L. (1983). "KIMMO: A Two-Level Morphological Analyzer," *Texas Linguistic Forum* 22:165–186.

Karttunen, L., and K. Wittenburg (1983). "A Two-Level Morphological Analysis of English," *Texas Linguistic Forum* 22:217–228.

Kay, M., and R. Kaplan (1982). "Word Recognition," unpublished draft ms. dated May 1982, Xerox Palo Alto Research Center, Palo Alto, California.

Lieber, R. (1980). *On the Organization of the Lexicon.* Ph.D. thesis, Department of Linguistics and Philosophy, M.I.T., Cambridge, Mass.

Lindstedt, J. (1984). "A Two-Level Description of Old Church Slavonic Morphology," *Scando-Slavica* 30:165–189.

McCarthy, J. J. (1982). "Prosodic Templates, Morphemic Templates, and Morphemic Tiers," in van der Hulst and Smith (1982a:191–223).

Nash, D. (1980). *Topics in Warlpiri Grammar.* Ph.D. thesis, Department of Linguistics and Philosophy, M.I.T., Cambridge, Mass.

Poser, W. (1982). "Phonological Representation and Action-At-A-Distance," in van der Hulst and Smith (1982b:121–158).

Swartz, S., ed. (1982). *Papers in Warlpiri Grammar in Memory of Lothar Jagst.* Work-Papers of SIL-AAB, Series A, Volume 6, Summer Institute of Linguistics, Berrimah, N.T.

Underhill, R. (1976). *Turkish Grammar.* Cambridge, Mass.: M.I.T. Press.

van der Hulst, H., and N. Smith, eds. (1982a). *The Structure of Phonological Representations, Part I.* Dordrecht, Holland: Foris Publications.

van der Hulst, H., and N. Smith, eds. (1982b). *The Structure of Phonological Representations, Part II.* Dordrecht, Holland: Foris Publications.